Information and Knowledge Management
ISSN 2224-5758 (Paper) ISSN 2224-896X (Online)
Vol.4, No.1, 2014
www.iiste.org

Requirements Modeling Methodology Based on Knowledge Engineering: A Case Study of Railway Control System

Nesrine Darragi , Simon Collart-Dutilleul, El-Miloudi El-Koursi
Univ Lille Nord de France, Ifsttar, Cosys-Estas
20 Rue Elisee Reclus, BP 70317, F-59666 Villeneuve D'ascq, France
Email: firstname.lastname@ifsttar.fr

Abstract
The complexity of the verification and the validation of embedded systems is increasing. This paper explores the first requirements engineering processes in the solution domain, which are analysis and specification. In this work we present an architecture of a requirement specification system. We show how the requirements are analysed and structured to generate a dependency graph. This latter will serve to analyse requirements and to model specifications on goal model. In this paper we will focus on the analysis, and structuring processes. We will explain the requirement classification criteria.
Keywords: Requirements Modeling, Qualification Strategy, Knowledge Engineering, Ontology, Dependency Graph, Embedded System, ERTMS/ETCS

1. Introduction

Requirements engineering "involves all life-cycle activities devoted to identification of user requirements, analysis of the requirements to derive additional requirements, documentation of the requirements as a specification, and validation of the documented requirements against user needs, as well as processes that support these activities"[19]. This definition lists the most important activities and refers to the qualifying process which covers dsigning and model verification.
Qualification is the evaluation of performance criteria. It combines verification and validation processes. The aim of qualification is to increase the level of reliability in the quality of the system, but also to improve the quality of other systems in the future.
Several methods are used to ensure the analysis of documents and their description as natural language processing (NLP), the use case diagrams [1] or scenarios [2]. These non-formal methods can be a source of ambiguity for analysts or system stakeholders. Formal methods, based on strictly mathematical languages, are mainly used to prove the consistency and completeness of requirements [3] and [4]. These last two features are the most difficult properties of requirements to verify [5].
We use the requirements of European Train Control System (ETCS) of European Rail Traffic Management System (ERTMS) as a case study. This project aims at a single train control system for the future transEuropean railway network. This is an example to illustrate how our methodology works on embedded systems in the railway domain. We use the specifications (Ref. 8.3.2) provided by the SUBSET SRS-6 V230. ERTMS/ETCS makes use of protocols and components which respect the European manufacturers consortium. Its requirement specifications are written in Natural Language and include diagrams and tables of dependencies between variables and transition conditions. We will focus only on some requirements of the Start of Mission (SoM) procedure of the European Vital Computer (EVC) which is the embedded system of ETCS.
In this paper, we first present a requirement qualification strategy to justify and argue our methodology for requirements specification. The methodology will be detailed in the third section. We then show in details each process and its application on requirements of embedded systems.

2. Requirements qualification strategy

According to the processes of requirements engineering, the proposed procedure, in terms of the new platform, covers the verification of stakeholders, the system, subsystems and component requirements. Figure 1 shows the requirements in various stages of development and the relationship between requirements in different levels and validation. Each stage of development is characterized by a specific process based on the nature of requirements.

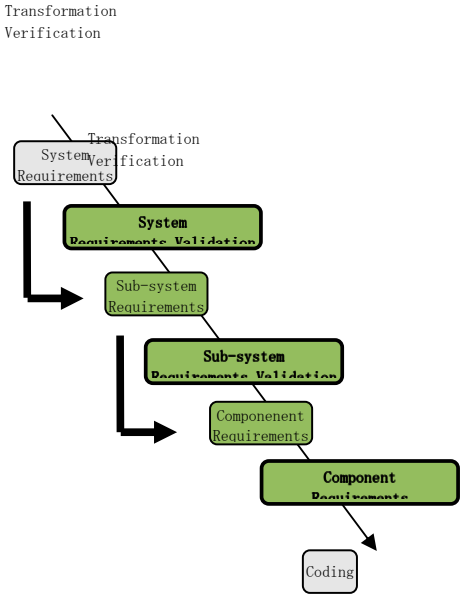

Figure 1: Scope of our qualification strategy

System validation may help to reduce risks, to prevent defects and to provide information for decision-making. Applying this process in the early stages of life requirements, is used to detect errors rather than facilitate the process of correcting and redefining the requirements. Another advantage of early validation is the re-usability of validated requirements. Furthermore, the requirements can be reused to qualify similar products.

Figure 1 portrays the insertion of supplementary tests (in bold) to eliminate defects as early as possible. We elaborate a test planning to define the objectives of validation and a central planning to verify the progress of testing itself. The stakeholder requirement validation concerns the verification of informal specification. Similarly, system requirement validation includes an analysis process of controlled natural language in order to detect anomalies. A component testing provides a very low level analysis. Qualification of component and requirement properties separately is not sufficient. We have to test the whole system at every level of integration, and even the refinement of requirements from coarse to fine granularity must be verified, which is shown in figure 1. During these tests a bi-directional traceability is created between low and high level requirements.

As figure 1 illustrates, the strategy focuses on the solution domain. The scope of our methodology is limited to verifying system requirements in natural language and the design of system components and its verifications in preparing the system deployment.

The requirement validation is an activity of the fundamental systems engineering. It consists of analyzing inputs in order to define exactly how the system must act. Besides of this goal, the analysis clarifies functional requirements, requirement qualities and design constraints determine a set of properties called SMART.

SMART is the abbreviation of Specific, Measurable, Attainable, Realizable and Time bounded or Traceable for T [6]. A requirement analysis is the process of studying the stakeholder needs. There are many requirement classifications. In our context, we distinguish between:

- Functional requirements as tasks, actions and activities of the system
- Performance requirements as Speed, accuracy, frequency, throughput
- Design requirements or external interface requirements (what to do)
- Design constraints (how doing it)
- Quality attributes as reliability portability supportability...

The author of [7] defines specificity as follows: “a requirement must say exactly what is required”. Therefore, a specification has to be clear without ambiguities, simple or atomic, of adequate requirements granularity (of one level) and consistent which means that used terms have unique sense. Other criteria of requirement specificity are the non-redundancy.

In the example 1, we show a “specificity” problem. In fact, the terms "these situations", "Stand-By mode", "Start of Mission" and “on-board” have to be defined before their uses. The term “previous situation” is not clear and needs precise explanations.

Example 1: Specificity Problem

The common point of all these situations is that the ERTMS/ETCS on-board is in Stand-By mode, but the Start of Mission will be different, since some data may be already stored on-board, depending on the previous situation

In the example 2, we awoke the atomicity problem. This is a composed requirement describing the

ERTMS/ETCS on-board equipment within different cases and describing the transition of the system status in different conditions. Beside of this first problem, we detect an ambiguity concerning the "train running number". "The possibility to enter/re-validate" and "depending on the status" must be clarified too.

Example 2: Atomicity Problem

Depending on the status of the Driver-ID, the ERTMS/ETCS on-board equipment shall request the driver to enter the Driver-ID(if the Driver-ID is unknown) or shall request the driver to revalidate or re-enter the Driver-ID(if the Driver-ID is invalid). The ERTMS/ETCS on-board equipment shall offer the driver the possibility to enter/re-validate(depending on the status) the Train running number. Once the Driver-ID and possibly the Train running number is/are entered or re-validated(E1), the process shall go to D2.

Typically, system developments have to be driven by SMART aspects. The validation of these properties has to be integrated into the life cycle of the software or the critical system. Our qualification strategy will be integrated on the specification methodology. The next section will describe its architecture and its functionalities.

3. Requirements specification system

We propose a requirement specification methodology for requirements structure and architecture determination. Figure 2 shows the overview of our methodology. This latter is implicated in the solution domain. The requirement elicitation is not considered as a process of our methodology.

We consider that we have a System Requirement Specifications noted SRS. We start by analysing requirements for complex embedded applications. We have guidelines to detect anomalies based on requirements analysis strategy. In case of analysis problem, we propose an edition of the requirement. The next process after the analysis is the structuring of requirements. We propose a structuring strategy based on a context-free grammar, a pattern base and knowledge base. This latters will be detailed in the next sections.

After requirements analysis for anomalies, our methodology analyses the structure of each requirement and determines the adequate pattern based on several algorithms for structure capturing and pattern construction.

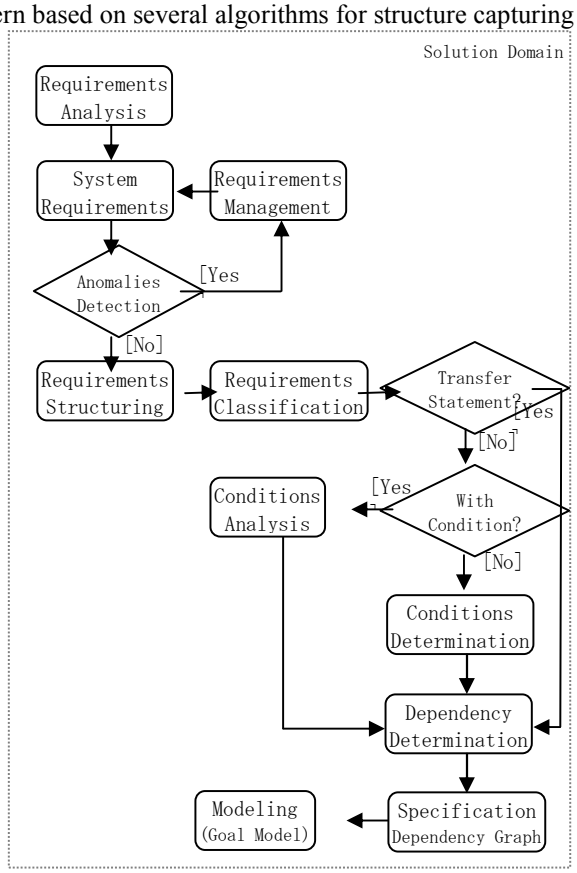

Figure 2: The overview of our methodology

3.1 Requirement Structuring Architecture

We propose a requirements structuring method largely building on cognitive techniques and shared domain knowledge for a sharp reasoning of each application domain and each case study.

Figure 3 shows the requirement structuring framework and how Knowledge models such as ontologies and glossaries may be connected with specification dependency graphs. The proposed structuring system includes

many algorithms which, based on extensible pattern and keyword bases, determine the structure and properties of requirements but also dependencies between them.
In this structuring strategy there is a need of system specific glossary suited to stakeholders, subsystems and functions collected from the system requirement specifications. We also require domain ontology specific to the system domain in order to instantiate glossary entries over a range of high level concepts and relationships. The figure 3 shows the relationship between system specific knowledge (SRS and glossary), the system domain knowledge (Domain Ontology) and the systems of specification elicitation and structuring.
Using structure in requirements engineering is explained by the fact that it may facilitate the understanding and the processing of a large amount of information, especially if the system is a large-scale distributed system such as an aircraft or railway control systems. The requirements structuring is a technique used in several works like [8] , [9], [10], [11] and [12] to ameliorate the expressiveness and to enable verification for more properties. Works of [13] and [14] focus specially on specification abstraction to elicit hiding information. [15], [16] and [17] are interested in the usability and the accessibility of specifications.

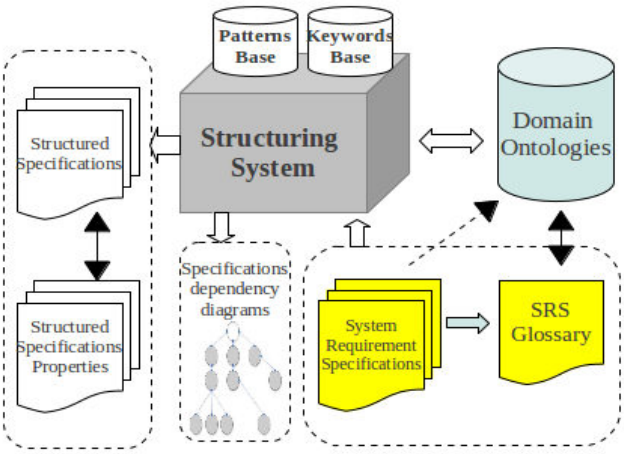

Figure 3: Requirement Structuring Framework

To express the context-free grammar of specifications we use a formal method called Extended Backus-Naur Form (EBNF). It is being well demonstrated in the linguistic literature that is difficult to capture natural languages in a context-free grammar.
In fact there are two problems in a formal grammar: generation of all outputs and parsing outputs. For example we may deal with two sentences with an identical semantic but with different structures. This is called a commutative sentence . Another problem is the ambiguity of sentences. In the case of a complex sentence, there are embedded phrases that replace the first sentence such as adjective-like or adverb-like phrases. Other studies [18] demonstrated that it is impossible to capture natural languages in a formal grammar.
What we propose in this section is the parse of a subset of English sentences with EBNF and not "all" English sentences. The subset is dedicated to specification declaration. In our context, a specification could be expressed as follows:
Specification:: = [Specification]+ |Declarative statement | Conditional statement | Command statement
Declarative statement:: = [Noun phrase]+ Verb phrase
Conditional statement:: = [Conditional keywords Declarative Statement]+
Command statement:: = [Conditional statement] Noun phrase Imperative keywords Transfert keyword Reference
Noun phrase::= [Proper noun Noun]+ [Verb]
Verb phrase::= Verb Noun phrase Adverb phrase
Adverb phrase:: = [Adverb]+
Imperative keywords:: = 'shall', 'must', 'is required to', 'will', 'should', 'is responsible for'
Conditional keywords:: = 'if', 'then', 'else', 'when','with', 'otherwise', 'only if'
Transfer keywords:: = 'go to', 'leads to'
The previous grammar will be necessary for the pattern discovery process. To acquire requirements structures, a syntactic comprehension is required. The example 3 shows how a requirement in natural language is processed to extract data and to determine the pattern which is conform to our grammar.
The pattern includes keywords from the base such as shall and although and concepts from the domain ontology

such as subsystem, condition, object.

3.2 Keywords Base

Our Keywords Base (KeywordsBase),is a table that contains words which determine different parts of the sentence such as fshall, while, every, if,...g. These indicators are identified by hand analysing and determining frequently used words and structures.We may identify many groups for these indicators based on its semantics.

We propose to extend the classification provided by [?] to cover the maximum of requirement structures. The following list indicates possible categories of keywords.

- Imperatives: shall, must, must not, is required to, are applicable, is responsible for, will, should,...
- Continuances: below, as follows, following, listed, in particular, support,...
- Directives: figure, table, for example, note, reference, see section, refer to, following {ref}, e.g,...
- Options: can, may, optionally,...
- Transfer: go to, leads to,...
- Conditionals: if, then, otherwise, once, in preparation, when, with, …
- Timed: immediately, later, at least, every, after, between, globally, not less than, exactly...
- Weak Phrases: adequate, as a minimum, if practical, as applicable, easy, as appropriate, be able to, be capable, but not limited to, capability of, effective, if practical, normal, provide for, timely, obvisouly, clearly, certainly, some, several, many, etc., and so on, such as, tubed,...
- Other Keywords: although, how, via, but, since, composed of not less than, from, of,...

Example 3: Structured Requirement

Requirement : The RBC shall inform the ERTMS/ETCS on board equipment that it accepts the train although the on-board has no "valid" position information.

Pattern: < Subsystem > shall < function >< object > [<Subsystem >< function >< object > although < condition >: [< Subsytem > has < state >]]

Data: [RBC,Inform,ERTMS/ETCS on board equipment, [RBC, accept, Train, [ERTMS/ETCS on board equipment, no valid position]]

3.3 *Guidance Ontology*

To automatically handle the semantics of our system, we need a description of the ontology presented by UML class diagrams. UML Profile is used to extend and customize UML models for a particular purpose of a particular domain.

According to Ontology Definition Metamodel (ODM) [20], a Profile UML is proposed in this part to describe our ontology which is an Ontology UML Profile (OUP) . Many works use UML and other software engineering techniques not only to develop ontologies in order to use existing advanced tools and standards but also to make ontologies understanding easier.

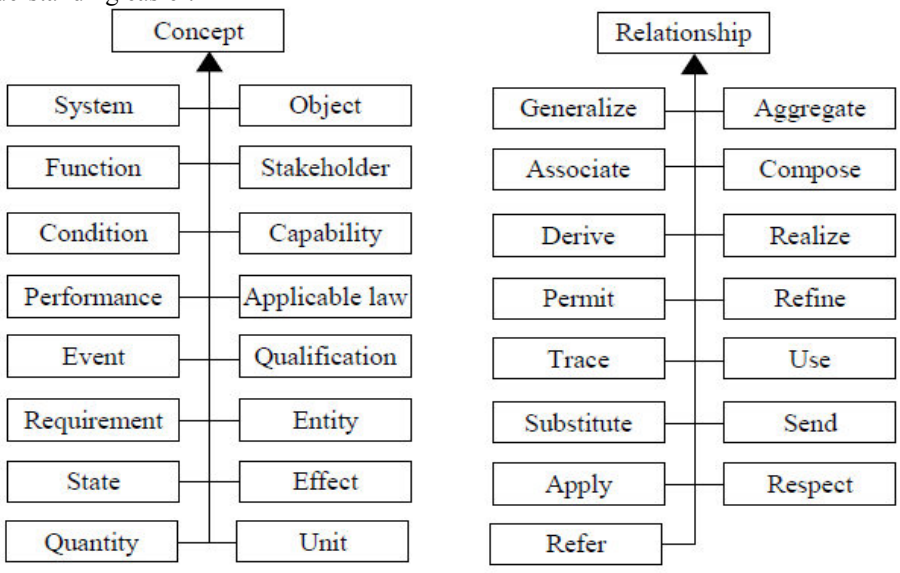

Figure 4: Concepts and properties of the domain ontology

In figure 4 we present a part of domain ontology. The first diagram defines the concepts and the second shows the relationships between different concepts called properties. We use UML class diagrams for the sake of simplicity, but also because we believe that it is sufficient to present our ontology,i.e. the concepts and relationships between them. We will include specific relations in UML as our list of properties (*Generalize, Aggregate, Associate, Compose*). This list is needed to show the dependencies between the different concepts represented as classes. In this work we will not show the specifications hierarchy but this relationships list will

be useful in the next step of our methodology. This ontology is specific to a domain so it must be generic so as to enable the hierarchical analysis. The list of properties is enhanced by a standard predefined UML dependency (*Derive, Realize, Permit, Refine, Trace, Use, Substitute, Send*).
We introduce a new property, called Respect. This is a property necessary in this context to show strong dependencies between requirements. It is used in cases where a requirement must follow a concept, definition, rule, standard or other requirement.
The ontology concepts in the field of requirements engineering, including safety requirements are defined in a general context. This allows us to represent the requirement specifications. Properties will be used not only to define the relationships between different concepts in the same requirement, but also to define the links between the requirements and traceability (See Fig. 5)

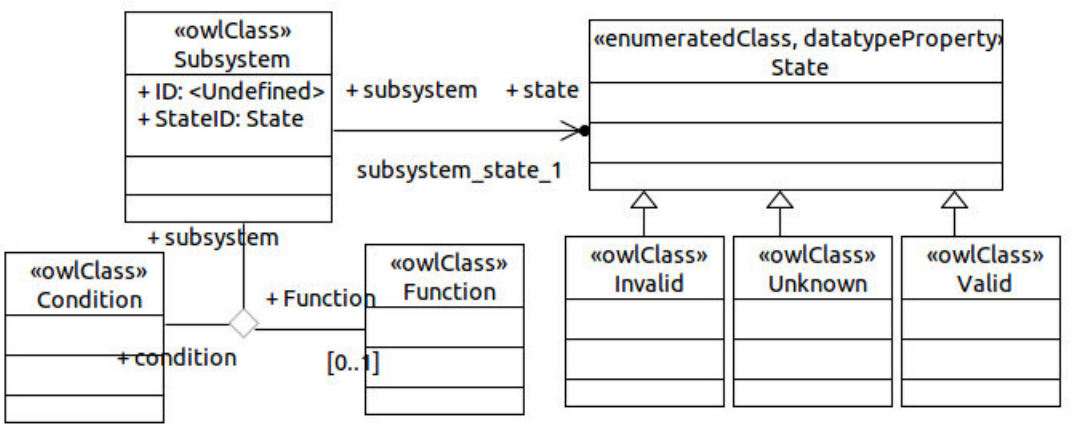

Figure 5: Part of the Guidance Ontology Meta-model

4. Requirements classification

The classification of structured requirements is not an easy process. In fact, we have to identify for each requirement to which category or cluster it belongs. There are several methods that try to solve this problem on the basis of data observation and training set called machine learning or by classification algorithms known as classifiers in data mining.
We begin by proposing different possible categories based on the nature of the requirement. We have two major categories which are statements with transfer indicators and statements without transfer keywords.
First of all, we have to define the term “transfer”. It means that a requirement has a transfer keyword as mentioned by the context-free grammar in the previous section. When we have a transfer statement, we can easily identify the dependencies between requirements and between components status. Each statement, describes the behaviors of a component or the constraints of transition between its status.
The goal of this classification is to facilitate the identification of dependencies between requirements. If there are indicators, it will be easy to process automatically the dependencies and to discover the requirements architecture. In the other cases, we need to define two sub-categories which are statements with conditions and without conditions. The first statements need to be syntactically analysed to determine the dependencies with other requirements based on information captured in conditions.
The processing of the second sub-category is the most difficult. We do not have explicit condition and without this information a semantic analysis is needed. In our case the problem of classification is limited to a set of requirements which does not contain any information about transitions. We are dealing with structured statements with identified parts and elements. The problem here consists only to identify implicit conditions, transition constraints and different component status.
To elicit hidden transition constraints from declarative statements we begin by classifying the list of requirements into “transfer” statement and “non-transfer” statement. For each element of the second category, if the templated requirement contains condition it is classified as “conditional” statement. The next step is to analyse the condition to identify the new component status. If the requirement does not contain any explicit information, we analyse the requirement semantically.

Example 4:

Specification: The ERTMS/ETCS on board equipment shall open the session with the RBC.
Pattern: < Subsystem > shall < function >< object >
Data: [ERTMS/ETCS on board equipment,Open, Session with RBC]
In contrast of the example 3, the example 4 shows a structured requirement which is neither a transfer statement nor a conditional one. To identify the dependency with another requirement, the “object “ and a “qualification”

of this latter have to be detected in another requirement condition.

5. Requirements specification system

5.1 Templated Specifications

In the first step of algorithm 1, we extract sentences from *SubRS* . Each sentence is split into different parts. If a list of sentence elements are not empty and the sentence parts are unitary, we may resume processing. This is what we referred to in the algorithm as ”understand” . If it is not possible to ”understand ” the sentence, we ask the assistance of a *SubRS* expert. The detectKeywords function uses a table of ”keywords” KeywordsBase . When keywords are detected, we try to determine the appropriate pattern by detectPattern .We use patterns to identify the structure of requirements. We proposed various patterns to handle *SubRS* by detecting keywords.

We analyse in the next phase of the same step the structure of sentence based on keywords and we classify it as a statement, a conditional sentence (*in case of if-then-else, when-then,...*) or a transfer sentence (*go to, leads to*). Finally, we apply a pattern from PatternBase or we create it if we are dealing with a new sentence structure. Example 5 shows a possible structure of a statement. Example 6 shows the structure of a transfer sentence. A conditional sentence is handled by pattern like the one shown in example 7.

This sentence categorization is not the unique or the best classification but we need to know information about dependencies between specifications and about the conditions of transitions between subsystem states.

The detection pattern callable unit shown by algorithm 3, returns a pattern of each statement after detecting instances of every statement element. DetectInstance uses *SubRS* glossary and places them on InstStack. All detected patterns are also placed in the StructSpec table.

Example 5. *Subsystem* ***shall*** *function object* ***every*** *performance units*

Example 6. Some Transfer patterns:

- *Subsystem* ***leads to*** *[Requirement Ref]*
- *Subsystem* ***go to*** *[Requirement Ref]*

Example 7. ***If*** *subsystem property is state , /****then*** *[*Declarative pattern *Or* transfer pattern*]*

If no pattern is detected, we create a new one based on its meta-model. The function is shown by algorithm 2. In this step, we are unable to identify real attributes such as the real subsystem or in the instance of state related to the property of the subsystem in example 7.

We need a type of knowledge base to help us to determine possible attributes of existing concepts. Patterns contain high level concepts that have to be instantiated for every SubRS in a lower level of abstraction. Structuring specifications leads to the separation of domain knowledge from operational knowledge. What we need is a finite list of terms such as a glossary. This will be a SubRS related glossary that heritates from a high level light ontology shown in Fig 5.

5.2 Specifications dependencies

The second part concerns the construction of a dependency graph from structured segments. This procedure translates the transfer statement (contains *go to or leads to* ..) into two nodes connected by directed edge from the source reference to destination reference . The translation of a structured specification which does not contain a transfer statement is more complicated. When specification is a conditional or declarative statement, a semantic analysis is necessary.

When the algorithm encounters external conditions in the statement, it creates the extcond node as well as the statement node and constructs control flow edges between them. An external condition is a conditional statement that refers to a <*Subsystem*> which is not belong to any of current subsystem stakeholders, unlike to internal condition, which does concern subsystem stakeholders. According to [21], a stakeholder is ”*an individual, group of people, organisation or other entity that has a direct or indirect interest or stake in a system* ”. To identify stakeholders of the *SubRS* , we need glossary entries.

If the algorithm encounters a conditional statementCS with many alternatives, it creates nodes for each one of them. We consider these alternatives as transition states and they have to be represented on a dependency graph in order to achieve the next state. Each state must to be satisfied.

Algorithm 1: Spec2CDG

```
Input: SubRS: Subsystem Requirement Specifications
Output: CDG: Control Dependence Graph of SubRS
Data: StructSpec: list of Structured Specifications;
ref: requirement reference;
Declare:
detectKeywords(S): detects keywords from KeywordBase in sentence S
detectPattern(S): detects pattern of sentence S from PatternBase
addNode(CDG,X): creates CDG node noted as X
addEdge(CDG,N1,N2): creates edge from node N1 to node N2 of CDG
Begin
while not at end of SubRS do
    read current;
    if understand then
        detectKeywords(current) ;
        if Keywords detected then
            detectPattern(current);
        else
            go back to the beginning of current section;
        if no pattern detected then
            create new pattern;
        save structured specification;
    else
        try to understand
for element of StructSpec do
    get ref from element;
    if element does not contain ref then
        determineDependencies(element,StructSpec);
    addNode(CDG,element);
    set currentNode element;
    if element contains many alternatives then
        for condition of alternatives do
            addNode(CDG,condition);
            addEdge(CDG,currentNode,lastNode);
        set currentNode lastNode;
    if element contains extern condition then
        get extcond;
        addNode(CDG,extcond);
        addEdge(CDG,currentNode,lastNode);
        set currentNode lastNode;
    get ref;
    if ref then
        addNode(CDG,ref);
        addEdge(CDG,currentNode,lastNode);
    else
        ask help;
End
```


Algorithm 3: detectPattern Procedure

Input: *S*: Sentence, *Ont*: Light Ontology
Output: *P*:A pattern of Sentence *S*
Data: *currentConcept*: A current anlysed concept
Concepts: Detected concepts list of *S*
InstStack: Detected instances list of Concepts
Declare:
detectConcepts(*Ont*,*S*): Detect concepts from *Ont* in sentence *S*
detectPossiblePatterns(*S*): Detect possible patterns for sentence *S* from PatternBase based on detected *keywords*
Begin
init Concepts;
init *InstStack*;
init *currentConcept*;
detectPossiblePatterns(*S*);
set Concepts by possible concepts;
while *not at the end of Concepts* **do**
  read *currentConcept*;
  detectInstance(*Ont*,*currentConcept*);
if *InstStack is not empty* **then**
  match *P* with Possible Patterns;
  if *P* **then**
    return *P*;
return Nil;

Algorithm 2: creationPattern Procedure

Input: *S*: Sentence, *Ont*: Light Ontology
Output: *P*:A pattern of Sentence *S*
Data: *currentConcept*: A current anlysed concept
Concepts: Detected concepts list of *S*
Inst: Detected instances list of Concepts
Declare:
detectConcepts(*Ont*,*S*): Detect concepts from *Ont* in sentence *S*
detectPossiblePatterns(*S*): Detect possible patterns for sentence *S* from PatternBase based on detected *keywords*
Begin
init Concepts;
init *Inst*;
init *currentConcept*;
detectPossiblePatterns(*S*);
set Concepts by possible concepts;
while *not at the end of Concepts* **do**
  read *currentConcept*;
  detectInstance(*Ont*,*currentConcept*);
if *Inst is not empty* **then**
  match *P* with Possible Patterns;
  if *P* **then**
    return *P*;
return Nil;

If there is no specification reference detected, determineDependencies is called. Based on similarity computing of different elements of StructSpec , the function returns specification references. This function makes use of various techniques of Natural Language Processing as syntactic and semantic processing. After obtaining the pattern and data, a dependence relationship must to be extracted from the specification. It is a complicated phase that needs rules and knowledge to be achieved. First of all, we have to define the term ”dependent on” and ”directly dependent on” .

Definition 1. A specification S_2 is dependent on specification S_1 (written S_1 δ S_2) if and only if:

- S_1 precedes S_2 in execution
- Execution of S_1 implies execution of S_2 in the future

Definition 2. A specification S_2 is directly dependent on specification S_1 (written S_1 δ^d S_2) if and only if:

- S_1 precedes S_2 in execution
- Execution of S_1 implies execution of S_2 in next step

When a specification contains ”transfer” keywords such as (*go to, leads to, ...*), it is possible to identify the dependence with other statements. Specifications without ”transfer” keywords are difficult to connect with other specifications. If the latter contains a conditional statement, this one could be an internal or external one. If it is an external condition, it will be translated as a node on dependency graph. The latter is directly dependent on current specification.

For all specifications which do not contain any condition, a semantic analysis is required. These specifications may be final states, initial states or statements which include implicit conditions. The elicitation and extraction of implicit data in general are quite a difficult operation, because there is a need for knowledge, rules and techniques.

To determine dependencies between all elements StructSpec, we propose some rules such as proposition 1 to facilitate the detection of dependencies. We assume that *DS* represents Declarative Statements. DS_τ are Declarative Statements with ”Transfer ” and $DS_{!\tau}$ are Declarative Statements without ”Transfer ” keyword.

CS are Conditional Statements. DS_τ U $DS_{!\tau}$ Í *DS* . A conditional statement is directly dependent on declarative statement with a condition which is expressed in 1.

Proposition 1. *If S_i is a declarative statement without ”Transfer” keyword, it exists conditional statement S_j that is directly dependent on S_i. [S_i Î $DS_{!\tau}$ Þ$ S_jƒÎ CS {S_j δ^d S_i}]*

We consider that a specification may be expressed as specificationºPattern+Data. In intuitionistic type theory, every term is annotated by its type, only welltyped terms are well-formed.

The previous example of structuring specification shows how to express by keywords, concepts and data a sentence. The pattern provides a syntactic description which contains predefined keywords and concepts. The list Data provides instances of concepts hence the importance of guidance ontology to determine hierarchical relationships between concepts and instances. **Although** or *{if, when}* keywords introduces a part of specification PRECONDITION . The POSTCONDITION is captured after shall or {to be, to have} keywords. The semantic of specification are captured by the structure itself. For example, when < function> has as instance ”Inform ”, we expect two objects for the statement: the receiever and the message. The latter may be a POSTCONDITION of the current specification. **has** or **is** in a condition could introduce states. The type of the next concept of statement could be determined by a dependent function types.

The algorithm 1 for constructing the dependency graph uses text as input and produces the control dependent Graph CDG. The complexity of algorithm 1 is O(n3) with n being a cost of unit operation.

Algorithm 4: determineDependencies Procedure

Input: *S*: Structured specification, *StrcutSpec*: List of structured specifications
Output: *Dep*:List of dependencies between *S* and elements of *StructSpec*
Data: *P*: A pattern
D: Data of a structured specification *S*
s: similarity percent
ε: a constant which means the threshold of acceptance of difference between two expressions
Declare:
similarity(e_1, e_2): Similarity percent between two elements e_1 and e_2.
Begin
P ← getPattern(*S*);
D ← getData(*S*);
get *objs*;
while *not at the end of objs* **do**
  get *currentobj*;
  for *element of StructSpec* **do**
    if *element is CS* **then**
      s ← similarity(*currentobj*, element);
      if $s \geq \varepsilon$ **then**
        D ← element;

6.

CONCLUSION

We proposed an approach for structuring requirements that makes use of a pattern base and a context-free grammar to structure and classify requirements. This methodology is integrating a qualification strategy that consists of incorporating validation processes for each step of requirement engineering. This intensive testing leads to more efficient development of complex systems.

REFERENCES

[1] Regnell B., Wesslen A. & Kimbler K. Improving the use case driven approach to requirements engineering. In 2nd IEEE International Symposium on RE'95 .1995;
[2] Yan Z., Jun H. & Xiaofeng Y. Scenario-driven component behavior derivation. Journal of Software .2007;
[3] Hu H. & Zhang L. Semantic-based requirements analysis and verication. In International Conference on electronics and Information Engineering. 2010;
[4] Xu Z. & Wu J. Ontology reasonong and services composition verication towards O-rgps requirement metamodel. In 3rd International Conference on Advanced Computer Theory and Engineering .2010;
[5] Davis A. M. Software Requirements: Analysis and Specification. Prentice Hall Press. 2nd edition . Upper Saddle River, NJ, USA.1993;
[6] Mike Mannion, Bary Keepence SMART Requirement Software Engineering ,Notes vol 20 N 2, Pages 42:47, April 1995;
[7] Mike Mannion, Bary Keepence Study Skills National Coaching Foundation ,1992;
[8] Dwyer M.B, Avrunin G.S & Corbett J.C Patterns in property specications for nite-state verication. In The 21st International Conference on Software Engineering ICSE .1999;
[9] Konrad S. & Cheng B. Real-time specication patterns. In the 27th International Conference of Software Engineering. ICSE .2005;
[10] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In CASSIS volume 3362, pages 49? 69. Springer-Verlag, LNCS, 2004;
[11] J. Berdine, C. Calcagno, and P. W. O'Hearn Smallfoot: Modular automatic assertion check- ing with separation logic In FMCO Springer LNCS 4111, pages 115-137, 2006;
[12] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. Software Tools for Technology Transfer 2005;
[13] M.J.Parkinson and G.M.Bierman. Separation logic and abstraction. In ACM POPL ,pages 247-258, 2005;
[14] P.W. O'Hearn, H. Yang, and J.C. Reynolds. Separation and Information Hiding. In ACM POPL , Venice, Italy, January 2004;
[15] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Multiple pre/post specications for heap- manipulating methods. In HASE pages 357?364, 2007;
[16] D. Distefano and M. J. Parkinson. jStar: Towards Practical Verication for Java. In OOPSLA 2008;
[17] G. T. Leavens and A. L. Baker. Enhancing the Pre- and Postcondition Technique for More Expressive Specications. In FM , September 1999;
[18] Shieber . Evidence Againt the context-freeness of natural language Linguistics and Philosophy , pages: 333-343, D.Reidel Publishing Company, 1985;
[19] DoD. Software technology strategy. December, 1991;
[20] OMG. Ontology Definition Metamodel. Version 1.0, Document Number: formal/2009-05-01;
[21] Elizabeth Hull, Ken JAckson and Jeremy Dick. Requirements Engineering. Spriger pages 7, 2004;